Fast Fourier Transform Implementation and Application on Major US cities' Temperature Data

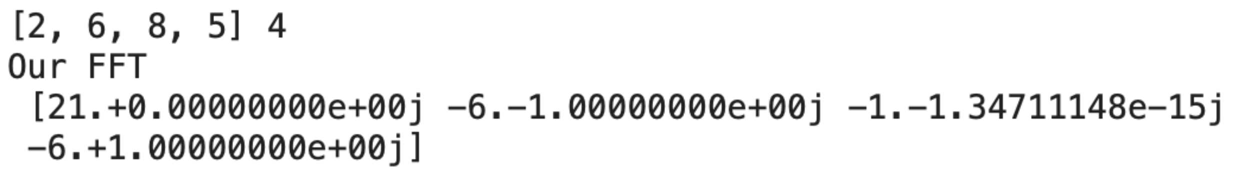
Yuting Ji, Scott Winters

Introduction

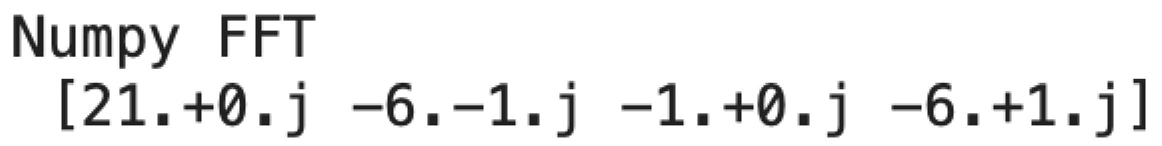
The Fast Fourier Transform (FFT) is an algorithm that computes the Discrete Fourier Transform (DFT) of a signal in O(nlogn) time. The algorithm to compute the DFT directly runs in O(n^2) time (Chugg, 2021). The DFT calculates a signal's frequency spectrum (ScienceDirect Topics). In other words, the DFT shows which frequencies occur in a signal. As such, in graphing the DFT, one can see which frequencies occur most often in a signal, a practical use of the DFT. As such, all of this can be done quicker with the FFT, considering its faster running time than the DFT. Although Python's Numpy package includes a built in FFT function, in this project, we implement the FFT function in Python, compare its results with the built in FFT function in MatLab, verify that it runs in O(nlogn) time, and use it to analyze average daily temperature data for six cities.

Implementing FFT Algorithm

To write our FFT function, we implemented an algorithm similar to Merge Sort, which runs in O(nlogn) time (GeeksforGeeks, 2022). We split the input array in half and apply our FFT function to each half recursively(similar to mergesort algorithm). When an input array, after being split in half multiple times, would finally have a length of 1, our FFT algorithm returns the input. Once both of the FFT recursive calls return their outputs, the np.concatenate combines the result from each recursive call back together and establishes the output array. This implementation allows the algorithm to run in O(nlogn) time, and when compared with Numpy's built in FFT function, gives the correct output, as shown below.

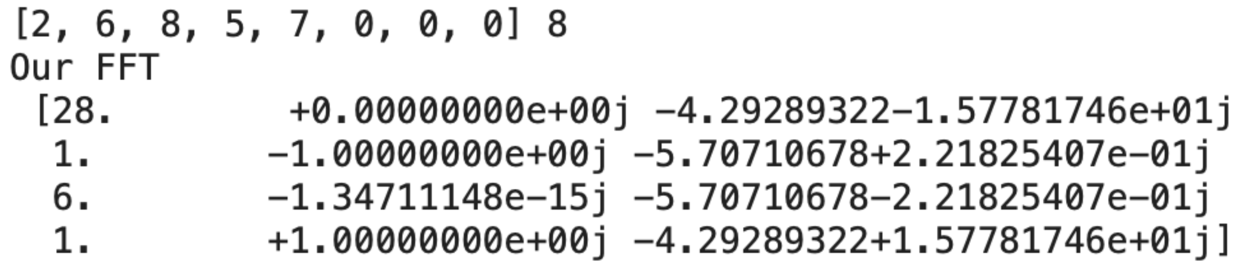


```
[2, 6, 8, 5] 4
Our FFT
 [21.+0.00000000e+00j -6.-1.00000000e+00j -1.-1.34711148e-15j
 -6.+1.00000000e+00j]
```

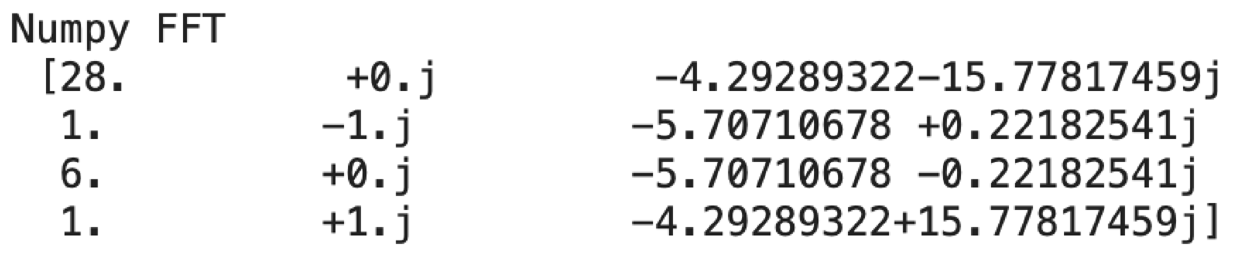


```
Numpy FFT
 [21.+0.j -6.-1.j -1.+0.j -6.+1.j]
```

However, there is one problem associated with our current algorithm: it can only take an input array that has length of a power of two. To solve this problem, we implemented a helper function, nearPowerOf2, to find the nearest power of two to the input array length. We then append zeros to the input array so that the updated array has a length of a power of two. After this step, we are then able to run our FFT function on an input of any size, with the correct output, as shown below.



```
[2, 6, 8, 5, 7, 0, 0, 0] 8
Our FFT
 [28.         +0.00000000e+00j -4.29289322-1.57781746e+01j
  1.        -1.00000000e+00j -5.70710678+2.21825407e-01j
  6.        -1.34711148e-15j -5.70710678-2.21825407e-01j
  1.        +1.00000000e+00j -4.29289322+1.57781746e+01j]
```

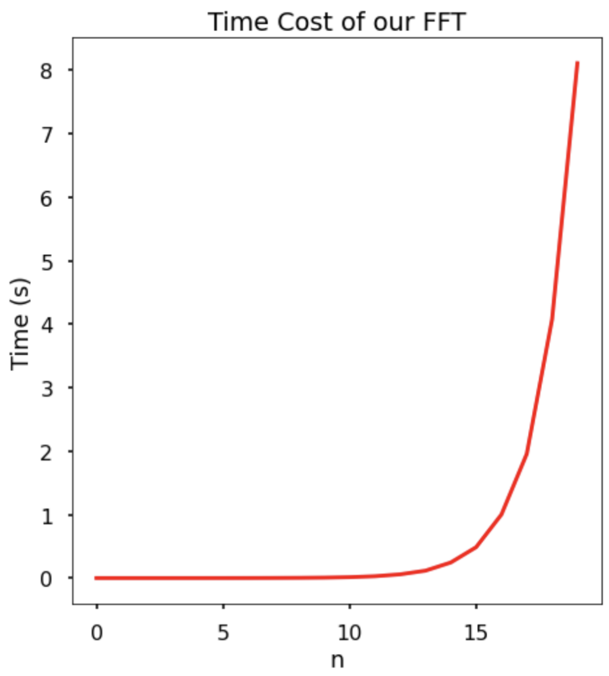


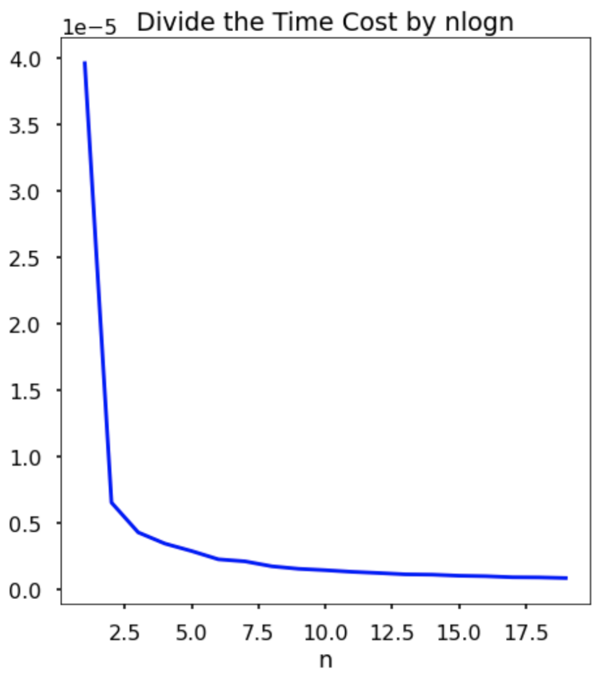
```
Numpy FFT
 [28.         +0.j          -4.29289322-15.77817459j
  1.         -1.j          -5.70710678 +0.22182541j
  6.         +0.j          -5.70710678 -0.22182541j
  1.         +1.j          -4.29289322+15.77817459j]
```

Time Analysis of FFT Algorithm

In order to time our fast fourier transform algorithm, we needed to time how long it takes to execute on inputs of various sizes. To do this, we imported Python's time package, which gave us the function time, which outputs the time when it is called, to use. In order to time our fast fourier transform algorithm, we created a new helper function, RunAndTime, which first called and saved the output of the time function, then called our fast fourier transform function with the same input, and then called and saved the output of the time function again. RunAndTime then subtracted the output of the first time call from the second, giving the time elapsed. Since the only action that occurred between the time calls was calling our fast fourier transform function, this time elapsed gives the time it takes our fast fourier transform function to execute. As such, RunAndTime then outputs a tuple consisting of the output of the fast fourier transform call and the time it took to execute.

Having defined a helper function to time our fast fourier transform algorithm, we then needed to call the helper function with various lengths of arrays as inputs, saving their time outputs in a list. We graphed said list, showing the execution time versus size of input n, shown below on the left. However, we needed to ensure that our fast fourier transform algorithm was running in O(nlogn) time. To do so, we divided the entire list of execution times by nlogn. If the algorithm was running in O(nlogn) time, then the new values in the list would be less than one for values of n approaching infinite. We then graphed the list, shown below to the right.

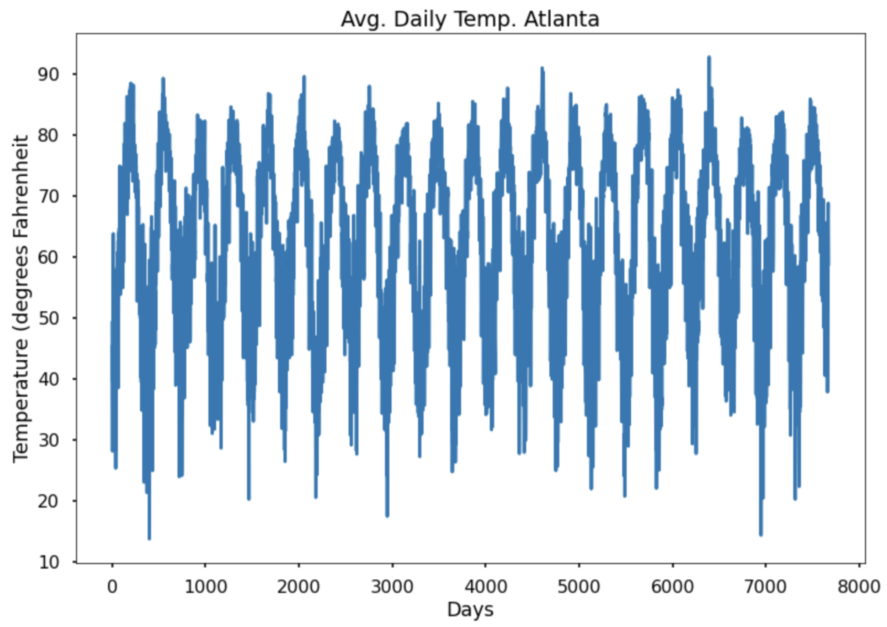


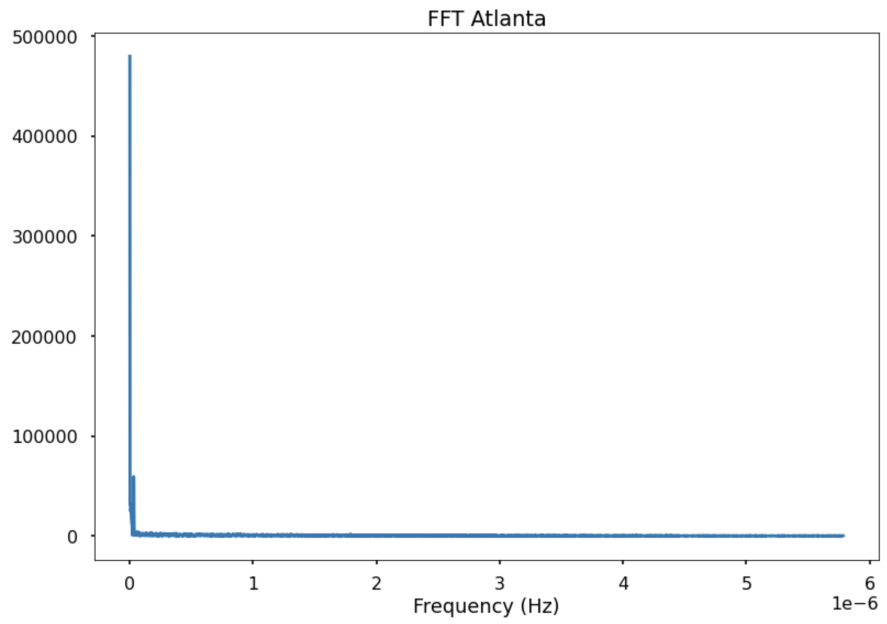


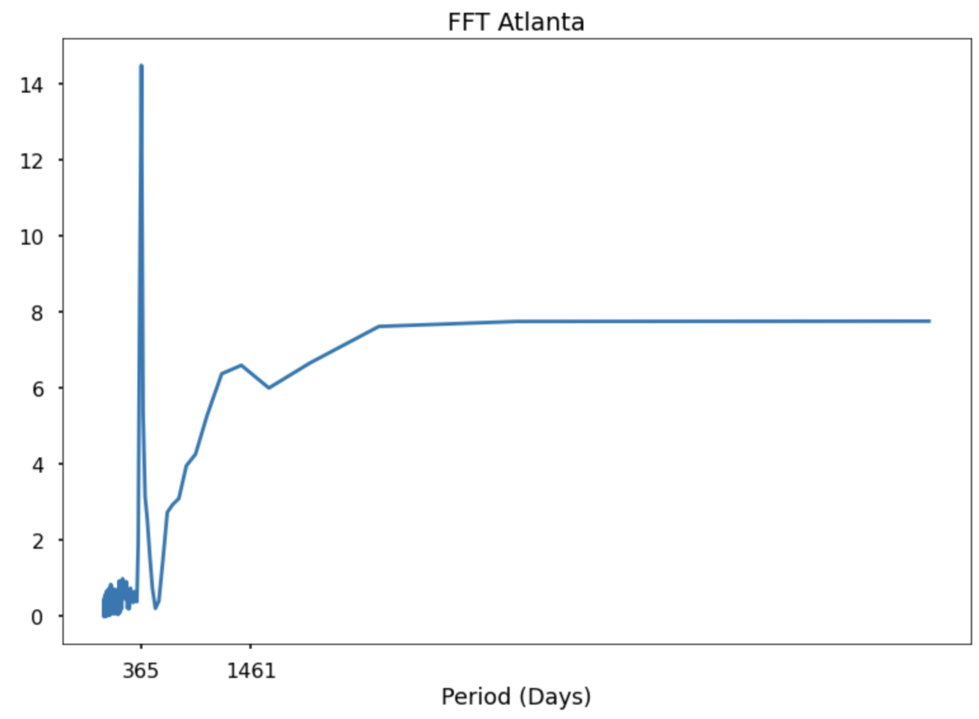
As one can see in the plot to the right, for all values of n greater than two, the time divided by nlogn is less than one. Furthermore, as n approaches infinity, the time asymptotically bounded, less than one. As such, we can conclude that our fast fourier transform algorithm does indeed run in O(nlogn) time.

Application of FFT Algorithm

Knowing that our fast fourier transform algorithm was running correctly and in O(nlogn) time, it was now time to apply our algorithm. To do so, we analyzed the data set “AvgDailyTermperature.mat” from the National Centers for Environmental Information, which includes average daily temperatures for each day from the last approximately 20 years (downloaded in 2016), for the six following cities: Atlanta, Boston, Miami, New York, Providence, and Washington DC (NCEI, 2016). Since the fast fourier transform shows the frequencies that are occurring most often, applied to this data, we could use our fast fourier transform algorithm to pick out any trends in the daily temperature data. So, to do this, first, we graphed the data for Atlanta. Next, we needed to convert the data from a numpy array to a list to use in our fast fourier transform function. Then, we applied to our fast fourier transform to the data for Atlanta and graphed it. However, this graph was not very useful, as the frequency was too low, leaving the peaks to lack definition. Therefore, we increased the frequency, and graphed this data again. These three plots are shown below.

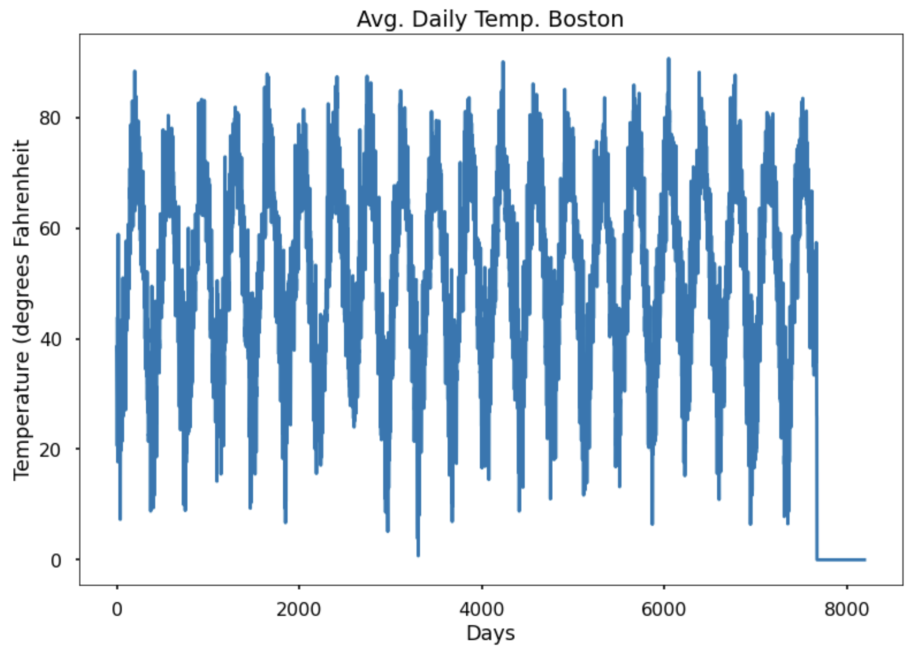


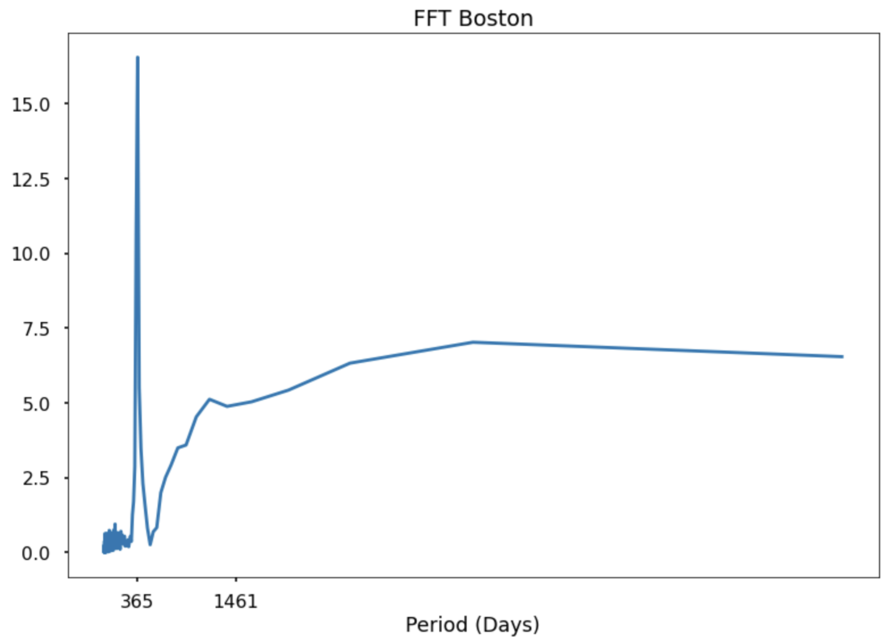


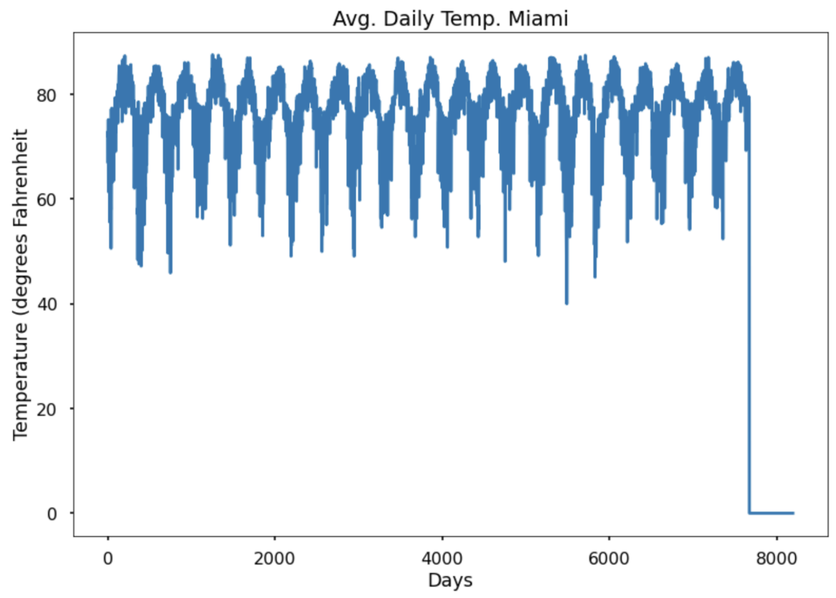


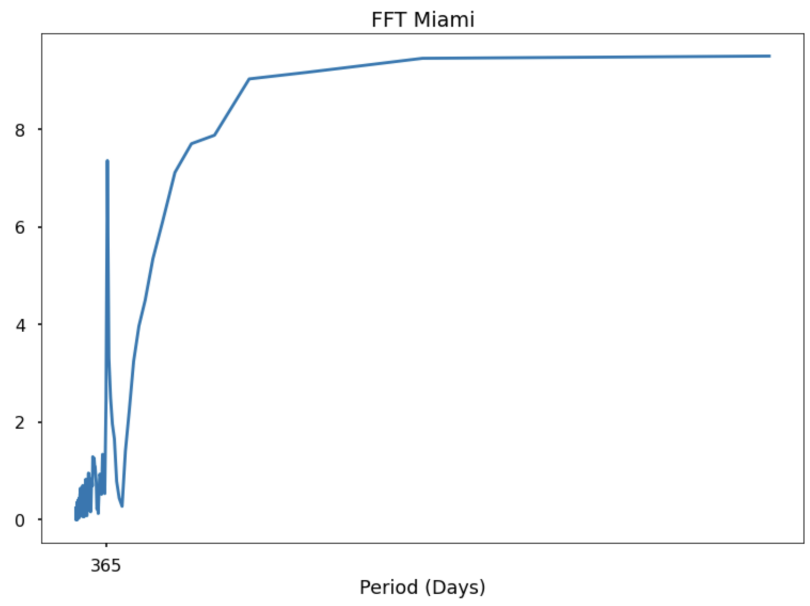
As shown above in the last plot, there are two peaks in the fast fourier transform of the temperature data from Atlanta. These are the frequencies at which signals repeat. The first peak is close to 365 days, while the second is close to 1461 days. The first peak, being at 365 days or 1 year, is significant, as it shows the repetitiveness of weather cycles year by year. In other words, the average temperatures tend to be the same at each time of the year every year. The second peak, being at 1461 days, which is equivalent to four years, is not as easily identified as significant. However, upon examining the first plot of the raw data, a similar trend can be discovered: about every four years, there are unseasonably high or low average temperatures at their highest and lowest points. Furthermore, in research, it can be found that certain weather patterns, such as La Nina that occurs every three to five years, can have effects like these (NOAA, 2015). As such, this fourth peak represents these extremes.

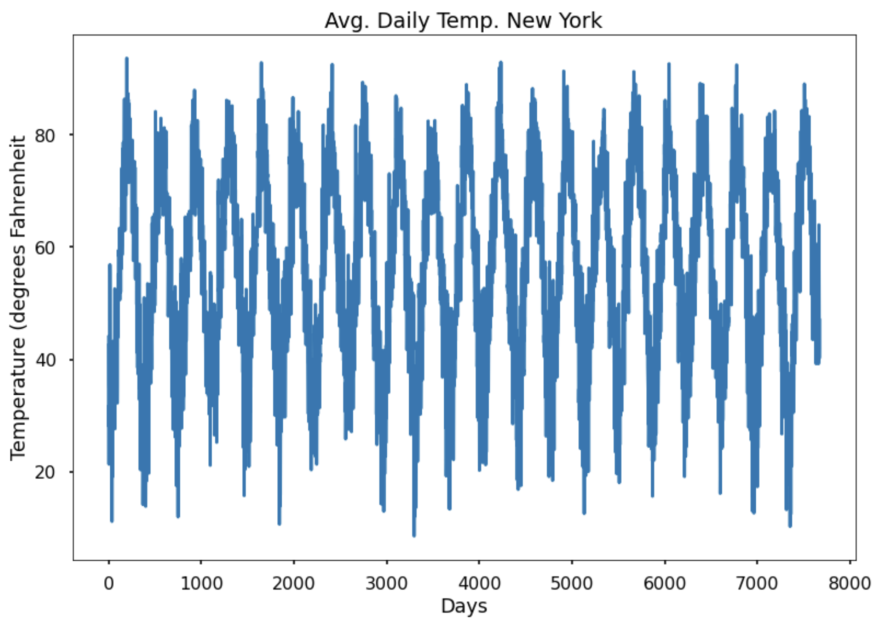
Shown below are the same plots (first and third, with the second not providing much value) for the five other cities in the dataset: Boston, Miami, New York, Providence, and Washington DC. Similar trends are seen.

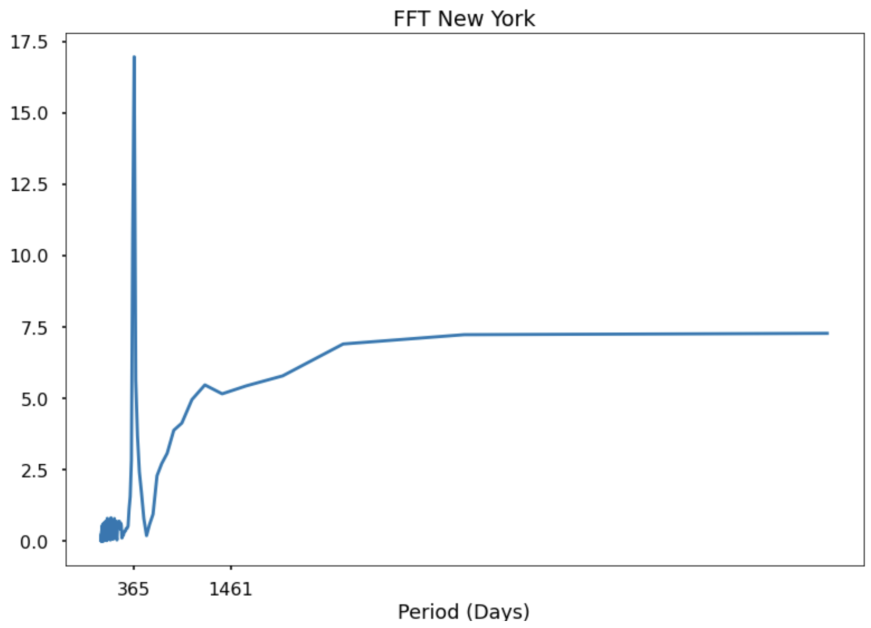


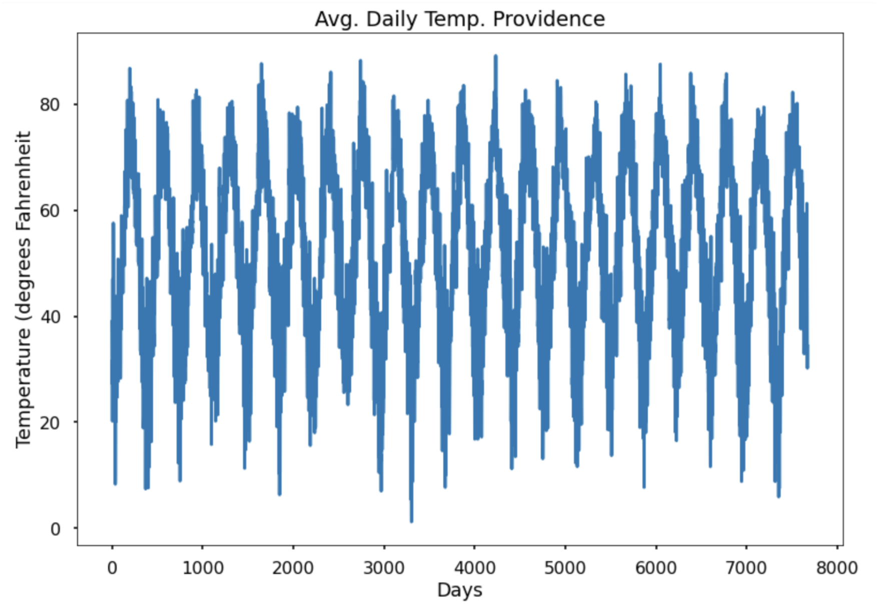


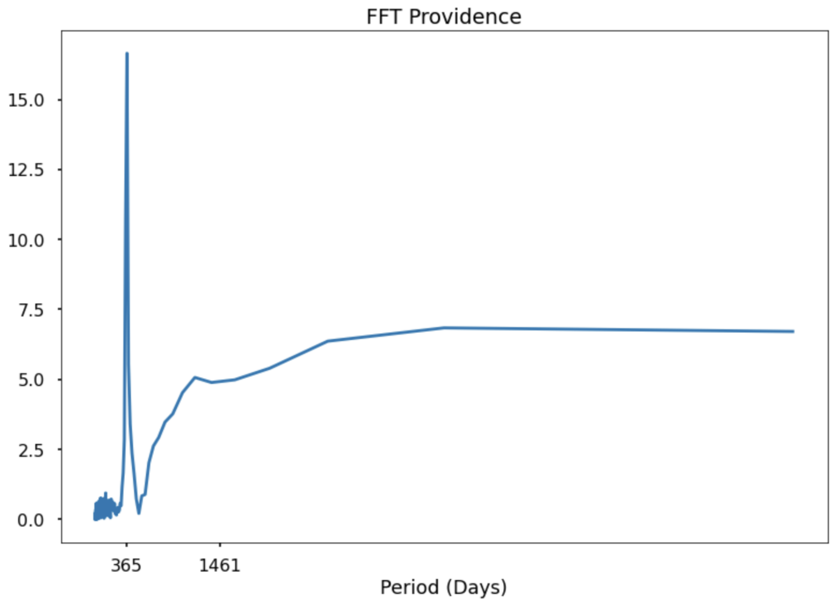

Avg. Daily Temp. Miami
Temperature (degrees Fahrenheit
0
20
40
60
80
0
2000
4000
6000
8000
Days

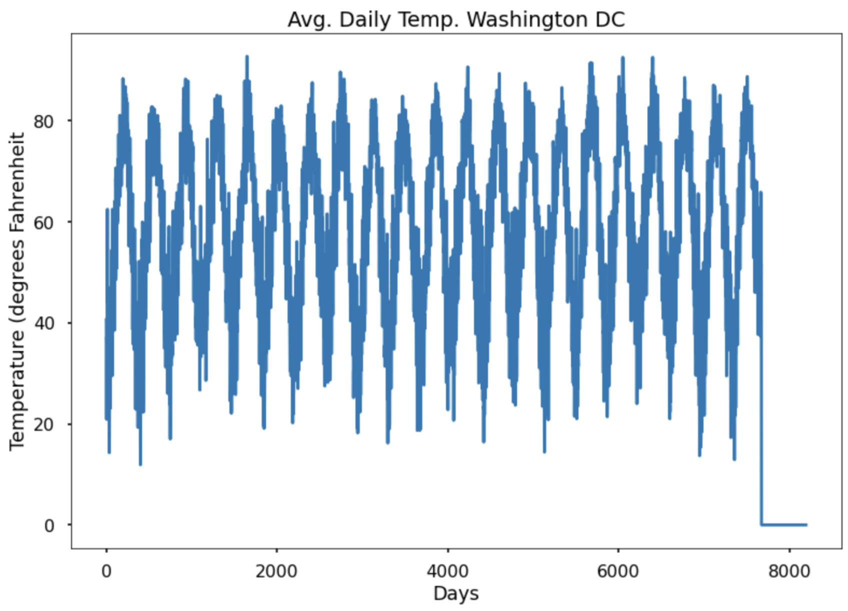

FFT Miami
0
2
4
6
8
365
Period (Days)

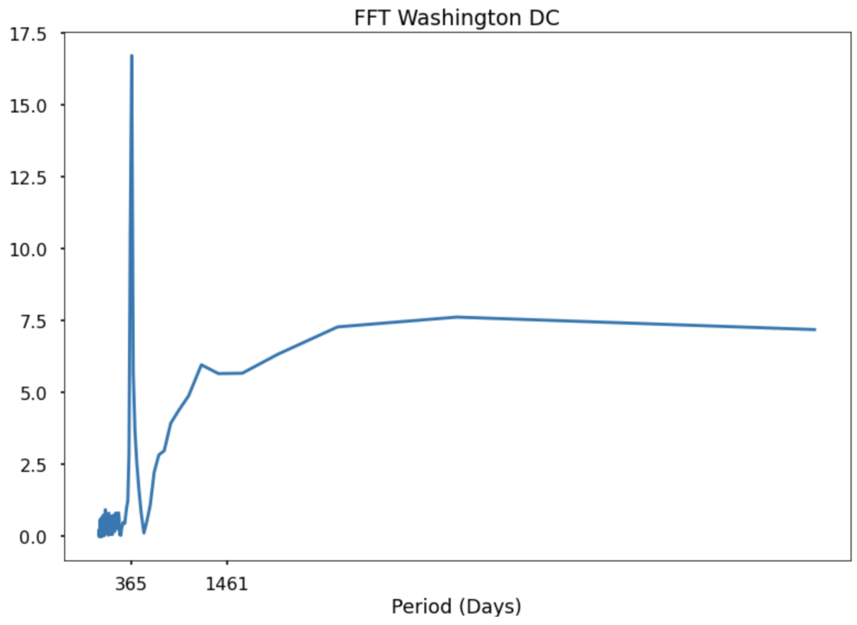

Avg. Daily Temp. New York
Temperature (degrees Fahrenheit
Days
0
1000
2000
3000
4000
5000
6000
7000
8000
20
40
60
80


FFT New York
Period (Days)
365
1461
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Avg. Daily Temp. Providence
Temperature (degrees Fahrenheit
Days
0
20
40
60
80
0
1000
2000
3000
4000
5000
6000
7000
8000


FFT Providence
Period (Days)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
365
1461





As seen in the plots above, similar trends, being peaks at 365 days and four years, are present in all the cities, except for Miami. Miami only has one peak at 365 days. It does not have a peak at four years, likely because Miami experiences more consistent weather than the rest of the United States, being in the tropics, and is thus less likely to have unseasonably high or low temperatures, which are expressed by the peaks in the plot of the fast fourier transform. Furthermore, upon examining the plot of the raw Miami temperature data, this theory is confirmed, as there is not as much variation in the height of the peaks as for the rest of the cities.

Conclusions

In summary, the FFT algorithm is a faster implementation of the DFT, and, when plotted, is extremely useful in visualizing the frequency spectrum of a signal. In our application to the average temperature dataset, it was quite useful in showing the seasonal patterns of weather, with repeating frequencies in temperatures every year and four years. Furthermore, our implementation of the FFT algorithm shows that the FFT can be written in relatively few lines of code by using recursion. We included a helper function to add zeros to the end of the array input, so as to ensure the input's length is a power of two. Therefore, for all input sizes, our FFT algorithm's outputs matched that of the built in Numpy FFT function in testing. Moreover, our FFT algorithm successfully ran in O(nlogn) time, saving significant time, as opposed to using the basic DFT algorithm on large inputs. In conclusion, the FFT algorithm is extremely useful in signal processing, and is an excellent tool in analyzing large datasets.

Contributions

Yuting wrote the FFT function, tested it against the Numpy FFT function, and did the execution time testing.

Scott did the average temperature dataset analysis, wrote most of the report paper, and most of the presentation for our report video.

References

Chugg, Keith M. *A Brief Introduction to the FFT - USC Hal Project*. 1 Apr. 2021, hal.usc.edu/chugg/docs/301/fft_intro.pdf.

“Discrete Fourier Transform.” *Discrete Fourier Transform - an Overview | ScienceDirect Topics*, www.sciencedirect.com/topics/engineering/discrete-fourier-transform.

“Merge Sort.” *GeeksforGeeks*, 22 Apr. 2022, www.geeksforgeeks.org/merge-sort/.

“Products.” *National Centers for Environmental Information (NCEI)*, 2016, www.ncei.noaa.gov/products.

US Department of Commerce, NOAA. “La Nina: How Does It Impact Our Winter Locally.” *La Nina: How Does It Impact Our Winter Locally*, NOAA's National Weather Service, 17 Mar. 2015, www.weather.gov/iwx/la_nina.